Oracle® JRockit

Performance Tuning Guide

Release R28

E15060-04

December 2011

Oracle JRockit Performance Tuning Guide, Release R28

E15060-04

Primary Author: Savija T.V.

Contents

Preface

This preface describes the document accessibility features and conventions used in this guide.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

About Profiling and Performance Tuning

This document describes the tuning techniques and options you can use to ensure your implementation of this JVM performs to maximum capabilities.

Tuning the Oracle JRockit JVM to achieve optimal application performance is a critical requirement for using this product. A poorly tuned JVM can result in slow transactions, long latencies, system freezes, and even system crashes.

Tuning is done as part of the system startup, by using various combinations of the start-up options described in *Oracle JRockit Command-Line Reference*.

The Oracle JRockit JDK provides the necessary tools to monitor your application at run time. When tuned correctly according to the recommendations in this document, the JVM runs smoothly and provides timely results. If run-time monitoring indicates problems, use the recommendations in this document to better tune the JVM.

This document contains information about the following topics:

- Tuning the JRockit JVM results in trade-offs among short garbage collection pause times, high application throughput, and low memory footprint. Chapter 2, "Understanding the Tuning Trade-offs" describes these trade-offs.
- Each Java application has its own behavior and its own requirements. The JRockit JVM can accommodate most of them automatically, but to get the optimal performance you must tune some basic parameters. Chapter 3, "Tuning the Oracle JRockit JVM: The First Steps" is an overview of the first steps you take when tuning the JRockit JVM and some best practices for tuning the JVM for a few Oracle applications.
- A correctly tuned memory management system minimizes the overhead caused by garbage collection and makes object allocation fast. Chapter 4, "Tuning the Memory Management System" describes the most important options available for tuning the memory management system in the JRockit JVM.
- The interaction among Java threads affects the performance of your application. Chapter 5, "Tuning Locks" contains information about the JRockit JVM options for tuning locks and describes how contention for locks are handled.
- Do you want your application to run smoothly with minimal pauses caused by the garbage collection? If the answer is yes, then tune for short pause times. Use the tuning techniques described in Chapter 6, "Tuning for Low Latencies" to ensure that pause times are kept to a minimum and transactions execute quickly.
- Do you want to minimize the total amount of CPU time spent in garbage collection and spend more time in the application layer? If the answer is yes, then tune for high application performance, or application *throughput*. Chapter 7, "Tuning for Better Application Throughput" describes how to tune JRockit JVM to

ensure that the Java application runs as fast as possible with minimal garbage collector overhead.

- For a stable performance of the JVM, you should analyze the performance variance and tune the JVM events as required. Use the tuning techniques described in Chapter 8, "Tuning for Stable Performance" to ensure a better performance.
- If you have limited memory resources, then tune the JVM for a small memory footprint by using the tuning techniques described in Chapter 9, "Tuning for a Small Memory Footprint".
- To decrease the JVM startup time for small applications, use the tuning techniques described in Chapter 10, "Tuning for a Faster JVM Startup".

2 Understanding the Tuning Trade-offs

This chapter describes how tuning the Oracle JRockit JVM leads to trade-offs among short garbage collection pause times, high application throughput, and low memory footprint.

This chapter contains the following topics:

- Section 2.1, "Pause Times and Throughput"
- Section 2.2, "Performance and Memory Footprint"

2.1 Pause Times and Throughput

With the JRockit JVM you can choose between short garbage collection pauses and maximum application throughput. The following sections describe why there is a trade-off.

2.1.1 Concurrent and Stop-the-World

The trade-off between garbage collection pauses and application throughput is partially caused by the mostly concurrent garbage collection strategy that enables short garbage collection pauses. No matter how efficient the garbage collection algorithm is that stops the Java threads during the entire garbage collection, a garbage collection algorithm that allows the Java threads to continue to run during parts of the garbage collection has shorter individual garbage collection pauses. A concurrent algorithm requires more work, because new objects are created and references between objects change during the garbage collection. All these changes must be kept track of, which causes overhead, and the garbage collector must make all the changes, which adds extra work. The more the garbage collector can do while the Java threads are paused, the less processing it has to do.

2.1.2 Compaction Pauses and Throughput

The mark and sweep garbage collection model can cause heap fragmentation when chunks of memory too small for allocated objects are freed between blocks of live objects. Compacting the heap reduces this fragmentation. Fragmentation has a negative impact on the overall throughput because it makes object allocation more difficult and garbage collections more frequent. The JRockit JVM does partial compaction of the heap in each garbage collection; this compaction occurs while all Java threads are paused. Moving objects takes time and compaction takes more time the more objects it moves. Reducing the amount of compaction shortens the compaction pause times, but it can reduce the overall throughput by allowing more fragmentation.

2.2 Performance and Memory Footprint

A small memory footprint is desirable for applications that run on machines with limited memory resources. There is a trade-off for having a small memory footprint and both application throughput and garbage collection pauses. The following sections describe some of the reasons for this trade-off.

2.2.1 Heap Size and Throughput

A large heap reduces the garbage collection frequency and fragmentation, improving the throughput of the application; however, a large heap increases the memory footprint of the Java process.

2.2.2 Bookkeeping and Pause Times

When you use a garbage collection mode that optimizes for short pauses, the Oracle JRockit JVM uses more advanced bookkeeping to track changes in the heap, references to compacted objects, and so on. All these increase the memory footprint. To tune the memory usage for bookkeeping, you must select a different garbage collection mode or strategy.

3 Tuning the Oracle JRockit JVM: The First Steps

This chapter describes the initial steps to tune the JRockit JVM and provides some best practices for tuning the JVM for a few different Oracle applications.

Every Java application has its own behavior and its own requirements. The Oracle JRockit JVM can accommodate many of the requirements automatically, but to derive optimal performance, you must tune some basic parameters. This chapter describes the following steps to tune the JVM:

- Section 3.1, "Step 1: Basic Tuning"
- Section 3.2, "Step 2: Performance Tuning"
- Section 3.3, "Step 3: Advanced Tuning"
- Section 3.4, "Best Practices"

For more information about tuning the JRockit JVM, see Chapter 4, "Tuning the Memory Management System" and Chapter 5, "Tuning Locks."

3.1 Step 1: Basic Tuning

Basic tuning consists of the following tasks:

- Tuning the Heap Size
- Tuning the Garbage Collection
- Tuning the Nursery Size
- Tuning the Pause Target

3.1.1 Tuning the Heap Size

The heap is where the Java objects are located. While a large heap requires fewer garbage collections, it can take longer to complete these collections. Typically, a heap should be at least twice the size of the live objects in the heap, meaning that at least half of the heap should be freed at each garbage collection. For server applications, you can usually set the heap as large as the available memory in your system allows, as long as this does not cause paging.

Use the following command-line options to set the heap size:

- `-Xms:`*`size`*: To set the initial and minimum heap size
- `-Xmx:`*`size`*: To set the maximum heap size

Example:

```
java -Xms:800m -Xmx:1000m MyServerApp
```

This command starts the JVM with a heap of 800 MB and allows the heap to grow up to 1000 MB.

For more information about setting the heap size, see Section 4.1.1, "Setting the Heap Size."

3.1.2 Tuning the Garbage Collection

Garbage collection reclaims space from objects that are no longer in use so that new objects can be allocated to this space. During garbage collection, system resources are used. By tuning the garbage collection, you can decide how and when the resources are used. The JRockit JVM has three garbage collection modes and a number of static garbage collection strategies. These enable you to tune the garbage collection to suit requirements of your application.

Select the garbage collection mode by using one of the following command-line options:

- `-Xgc:throughput` optimizes garbage collection for application throughput.
- `-Xgc:pausetime` optimizes garbage collection for short garbage collection pauses.
- `-Xgc:deterministic` optimizes garbage collection for very short and predetermined garbage collection pauses.

  The `-Xgc:deterministic` option is available only as part of Oracle JRockit Real Time (for more information about Oracle JRockit Real Time, see *Oracle JRockit Real Time Introduction*)

You might start a transaction-based application that requires low latencies with the following command:

```
java -Xgc:pauseTime MyTransactionApp
```

This command starts the JVM with garbage collection optimized for short garbage collection pauses.

For more information about selecting a garbage collection mode or a static garbage collection strategy, see Section 4.2, "Selecting and Tuning a Garbage Collector."

3.1.3 Tuning the Nursery Size

Some of the garbage collection modes and strategies in the JRockit JVM use a nursery. The nursery is an area of the heap where new objects are allocated. When the nursery becomes full it is garbage collected separately in a young collection. The frequency and duration of young collections is determined by the nursery size. A large nursery decreases the frequency but the duration of each young collection might increase.

The nursery size is adjusted automatically to optimize for application throughput if you use the `-Xgc:throughput` or `-Xgc:genpar` option. Typically, the nursery size should be as large as possible while maintaining short young collection pauses. Depending on the application, a nursery size can be anything from a few megabytes up to half of the heap size.

Set the nursery size by using the `-Xns:`*`size`* command-line option.

Example:

```
java -Xms:800m -Xmx:1000m -Xgc:pausetime -Xns:100m MyTransactionApp
```

This command starts the JVM with a heap of 800 MB, allowing it to grow up to 1000 MB. The garbage collection is set to optimize for pause times, and the nursery size is set to 100 MB.

For more information about tuning the nursery size, see Section 4.1.2, "Setting the Nursery and Keep Area Size."

3.1.4 Tuning the Pause Target

The `-Xgc:pausetime` and `-Xgc:deterministic` command-line options use a pause target for optimizing the pause times while keeping the application throughput as high as possible. A higher pause target usually allows for a higher application throughput; therefore, set the pause target as high as your application can tolerate.

Set the pause target by using the `-XpauseTarget:`*`time`* command-line option.

For example, a transaction-based application with transactions that usually take 100 milliseconds and time out after 400 milliseconds is started with the following settings:

```
java -Xgc:pausetime -XpauseTarget:250ms MyTransactionApp
```

The JVM starts with garbage collection optimized for short pauses with a pause target of 250 milliseconds. This leaves a 50 milliseconds margin before time-out for 100 milliseconds transactions that are interrupted by a 250 milliseconds garbage collection pause.

For more information about tuning the pause target, see Section 4.2.1.2.1, "Setting a Pause Target for Pausetime Mode."

3.2 Step 2: Performance Tuning

To tune your JVM for better application throughput, you must first assess the throughput. A common way of measuring the application throughput is to time the execution of a predefined set of test cases. Optimally, the test cases must simulate different use cases and they must be as close to real scenarios as possible. Also, one test run should take at least a few minutes, so that the JVM has time to warm up.

The following sections describe a few optional performance features that could improve the performance for many applications.

- Call Profiling
- Large Pages

3.2.1 Call Profiling

Call profiling enables the use of more advanced profiling for code optimizations and can increase the performance for many applications.

To use this feature on your application, add the following option to the Java command line:

```
-XX:+UseCallProfiling
```

For more information about the `-XX:(+|-)UseCallProfiling` option, see the *Oracle JRockit Command-Line Reference*.

3.2.2 Large Pages

A large page is essentially a block of contiguous physical-memory addresses that are reserved for a process.

Benefits

Large pages help in improving the performance of applications that require a lot of memory and access memory frequently.

When a process seeks data from memory, the processor looks up the translation look-aside buffer (TLB) — a cache of recently used virtual-to-physical address space translations stored in the processor memory — to find out the physical addresses (RAM or hard disk) that hold the require data. When large pages are enabled, a single entry in the TLB could represent a large contiguous address space, potentially reducing the TLB-lookup frequency and improving performance. In addition, when large pages in physical memory are reserved for a memory-intensive process, the CPU needs to track a small number of pages: for example, only 1024 large (2 MB) pages for a process running with 2 GB of heap, as opposed to 524288 small (4 KB) pages; so it is more likely that the virtual-to-physical address translation for the required data exists in the TLB, without which the processor would need to resort frequently to the slower method of reading the hierarchical page table stored in memory.

Drawback

When a large portion of physical memory is reserved for a process, other applications running on the system might encounter excessive paging (data swapped between hard disk and RAM), affecting the overall performance of the system.

Prerequisite for Using large pages

Support for large pages should be available and enabled in your operating system. The use of large memory pages is enabled by default on Solaris, but needs to be explicitly enabled on Windows and Linux systems.

For information about how to enable large pages on specific operating systems, see `http://java.sun.com/javase/technologies/hotspot/largememory.jsp` and the documentation for your operating system (Windows: `http://technet.microsoft.com/en-us/library/ms190730.aspx`; Linux: see the `vm/hugetlbpage.txt` file in the kernel documentation).

Note: Even with operating systems in which the support for large pages is available and enabled, at times, the JVM might not be able to reserve large pages because the pages might not be contiguous. To ensure that the JVM can use large pages, you must enable large pages in the operating system as soon as possible after the system starts (possibly, by using a startup script); otherwise, ongoing memory usage can cause fragmentation leading to reduction in the number of available contiguous large pages. Rebooting the system might solve this problem.

Procedure for Enabling Large Pages in the JRockit JVM

When running the JRockit JVM on operating systems in which large pages are supported, you can enable large pages by using the `-XX:+UseLargePagesFor[Heap|Code]` option. For more information, see the *Oracle JRockit Command-Line Reference*.

3.3 Step 3: Advanced Tuning

Some applications benefit from further tuning. Verify the results of the tuning by monitoring and benchmarking your application. Advanced tuning of the JRockit JVM, if done correctly, could improve the performance and predictable behavior; incorrect tuning can cause uneven performance, low performance, or performance degradation over time.

This section describes the following topics:

- Tuning Compaction
- Tuning the Thread Local Area Size

3.3.1 Tuning Compaction

Compaction moves objects closer to each other in the heap, reducing the fragmentation and making object allocation easier for the JVM. The JRockit JVM compacts part of the heap at each garbage collection (or old collection, if the garbage collector is generational).

Compaction can cause long garbage collection pauses. To assess the impact compaction has on garbage collection pauses, you can monitor the `-Xverbose:compaction` or `-Xverbose:gcpause` outputs or create a recording with the JRockit Flight Recorder and examine the garbage collection pauses in that recording. Look for old collection pause times and pause parts called compaction and reference updates. The compaction pause times depend on the compaction ratio and the maximum references. For more information about JRockit Flight Recorder, see the *Oracle JRockit Flight Recorder Run Time Guide*.

3.3.1.1 Compaction Ratio

The compaction ratio determines the percent of the heap to be compacted during each old collection.

You can specify the compaction ratio by using the following option:

```
-XXcompaction:percentage=percentage
```

If the compaction causes undesirably long garbage collections, tune the compaction ratio. First, lower the compaction ratio to 1.

- If the garbage collection time returns to expected levels, gradually increase the compaction ratio until the garbage collection time becomes too long. A good value for the compaction ratio is usually between 1 and 20.
- If the garbage collection time is long even after you set the compaction ratio to 1, change the maximum references as described in Section 3.3.1.2.

Setting the compaction ratio too low might increase the fragmentation and dark matter (that is, free space too small to be used for object allocation). You can see the amount of dark matter in the JRockit Flight Recorder recordings. For more information, see the *Oracle JRockit Flight Recorder Run Time Guide*.

3.3.1.2 Maximum References

By setting maximum references, you can define a limit that restricts the number of references that can be made to objects within a compaction area. If the number of references exceeds this limit, the compaction is canceled.

You can define the maximum references by using the following option:

```
-XXcompaction:maxReferences=value
```

If the compaction causes undesirably long garbage collections, tune the maximum references. First, set the maximum references as low as 10000. If that solves the problem, gradually increase the maximum references until the compaction times become too long. A typical value for the maximum references is between 100000 and several million. Lower values are used when the pause time limits are very low.

Setting the maximum references too low can stop the compaction altogether. This appears in the verbose logs or in a JRockit Flight Recorder recording as a "skipped" compaction. Running without any compaction at all can lead to increased fragmentation, which, in the end, forces the JVM to perform a full compaction of the entire heap at once. This can take several seconds. Oracle recommends that you do not decrease the maximum references unless absolutely necessary.

Note: The `-XXcompaction:maxReferences` option has no effect when the `-Xgc:deterministic` or `-Xgc:pausetime` options are used. For these garbage collection modes, do not tune the compaction manually; use the `-XpauseTarget` option to tune the garbage collection pauses.

For more information, see Section 4.3, "Tuning Compaction."

3.3.2 Tuning the Thread Local Area Size

Increasing the Thread Local Area (TLA) size is beneficial for multi-threaded applications where each thread allocates a lot of objects. Increasing the TLA size is also beneficial when the average size of the allocated objects is large because this allows larger objects to be allocated in the TLAs; however, increasing the TLA size too much can cause more fragmentation and more frequent garbage collections. To assess the sizes of the objects allocated by your application, create a JRockit Flight Recorder recording and view object allocation statistics in the JRockit Flight Recorder. For more information about JRockit Flight Recorder, see *Oracle JRockit Flight Recorder Run Time Guide*.

The TLA size is set using the following option:

```
-XXtlaSize:min=size,preferred=size
```

The `min` value is the minimum TLA size, while the `preferred` value is a preferred size. This means that TLAs will be the `preferred` size whenever possible, but can be as small as the `min` size. Typically, the preferred TLA size can be up to twice the size of the largest commonly used object in the application. Adjusting the minimum size can affect the garbage collection performance, but it is seldom necessary. A typical value for the minimum size is 2 KB.

For more information about tuning the TLA size, see Section 4.4, "Optimizing Memory Allocation Performance."

3.4 Best Practices

This section describes best practices for tuning the JRockit JVM for the following specific applications:

- Oracle WebLogic Server
- Oracle WebLogic SIP Server

- Oracle Complex Event Processing
- Eclipse IDE
- Utility Applications
- Batch Runs

3.4.1 Oracle WebLogic Server

Oracle WebLogic Server is an application server and requires high application throughput. An application server is often set up in a controlled environment on a dedicated machine. Do the following when tuning the JRockit JVM for Oracle WebLogic Server:

- Use a large heap, up to several gigabytes if the system allows for it.
- Set the initial or minimum heap size (`-Xms`) to the same value as the maximum heap size (`-Xmx`).
- Use the default garbage collection mode: `-Xgc:throughput`

3.4.2 Oracle WebLogic SIP Server

Oracle WebLogic SIP Server is an application server designed for the communications industry. Typically it requires fairly low latencies and is run in a controlled environment on a dedicated machine. Do the following when tuning the JRockit JVM for Oracle WebLogic SIP Server:

- Use a large heap, at least 2 GB if the system allows for it.
- Set the initial or minimum heap size (`-Xms`) to the same value as the maximum heap size (`-Xmx`).
- Use the garbage collection mode optimized for pause times (`-Xgc:pausetime)` or the static generational concurrent garbage collector (`-Xgc:gencon).`
- Use a small nursery, in the range of 50 MB to 100 MB.
- Decrease the compaction ratio or maximum references to lower and even-out the compaction pause times. For more information, see Section 3.3.1, "Tuning Compaction."

3.4.3 Oracle Complex Event Processing

Oracle Complex Event Processing is used for applications that are based on an event-driven architecture. Typically, it requires very low latencies and is run in a controlled environment on a dedicated machine. Do the following when tuning the JRockit JVM for Oracle Complex Event Processing:

- Use a heap size of 1 GB to fully utilize the deterministic garbage collection mode.
- Set the initial or minimum heap size (`-Xms`) to the same value as the maximum heap size (`-Xmx`).
- Use the garbage collection mode optimized for low and deterministic latencies (`-Xgc:deterministic).` The deterministic garbage collection mode is only available as a part of Oracle JRockit Real Time.

3.4.4 Eclipse IDE

Eclipse IDE requires fast response times and it is typically run on a workstation together with many other applications. You can also use Oracle JRockit Mission Control as an Eclipse IDE plug-in. For more information about integrating Oracle JRockit Mission Control with an Eclipse toolkit, see *Oracle JRockit Mission Control Client Mission Control Introduction*.

Do the following when tuning the JRockit JVM for an Eclipse IDE:

- Use a maximum heap size that is lower than the amount of RAM in the system and leaves space for the operating system and a varying number of other applications running simultaneously.
- Set the initial or minimum heap size (`-Xms`) lower than the maximum heap size (`-Xmx`) to allow the JVM to resize the heap when necessary.
- Use the garbage collection mode optimized for short pauses (`-Xgc:pausetime`) or the default garbage collection mode (`-Xgc:throughput`).

3.4.5 Utility Applications

Java utility applications that run for a short time and have a simple and specific purpose (for example, `javac`), require a fast startup and often do not need a lot of memory. To tune the JRockit JVM for this kind of application, do the following:

- Use a small heap, anything from 16 MB and up, depending on the requirements of the application.
- Set the initial or minimum heap size (`-Xms`) to the same value as the maximum heap size (`-Xmx`).
- Use the default garbage collection mode: `-Xgc:throughput`.

3.4.6 Batch Runs

Applications that process large batches of data (for example, applications for XML processing or data mining) require maximum application throughput but are seldom sensitive to long latencies. To tune the Oracle JRockit JVM for such applications, do the following:

- Set the heap size as large as possible, while leaving some physical memory for the operating system and other applications that might be running at the same time.
- Set the initial or minimum heap size (`-Xms`) to the same value as the maximum heap size (`-Xmx`).
- Use the default garbage collection mode: `-Xgc:throughput`.
- Increase the TLA size. For more information, see Section 3.3.2, "Tuning the Thread Local Area Size."

4 Tuning the Memory Management System

This chapter describes the most important options available for tuning the memory management system in the JVM.

Memory management is about allocation of objects. One part of the memory management system finds a free spot for a new object, while another part garbage collects old objects to create even more free space for new objects. The more objects a Java application allocates, the more resources are required for memory management. When correctly tuned, a memory management system minimizes the overhead caused by garbage collection and makes object allocation fast.

This chapter includes information about the following topics:

- Section 4.1, "Setting the Heap and Nursery Size"
- Section 4.2, "Selecting and Tuning a Garbage Collector"
- Section 4.3, "Tuning Compaction"
- Section 4.4, "Optimizing Memory Allocation Performance"

For more information about memory management in the Oracle JRockit JVM, see the "Understanding Memory Management" section in *Oracle JRockit Introduction to the JDK*.

4.1 Setting the Heap and Nursery Size

The heap is the area where the Java objects reside. The heap size has an impact on the performance of the JVM and the Java application.

When the JVM uses a generational garbage collection strategy, part of the heap is reserved for the nursery. All small objects are allocated in the nursery (young space). When the nursery becomes full, a young collection is performed, where objects that have been active in the nursery are moved to old space, which is the rest of the heap.

To distinguish between recently allocated objects and objects that have been in the nursery for a while, the JVM uses a keep area. The keep area contains the most recently allocated objects in the nursery and is not garbage collected until the next young collection.

4.1.1 Setting the Heap Size

Command-line options: `-Xms:`*`size`* `-Xmx:`*`size`*

The heap size affects allocation speed, garbage collection frequency and garbage collection times. A small heap fills quickly and must be garbage collected more often. A large heap results in overhead at garbage collection times. A heap that is larger than the available physical memory in the system must be paged out to disk, which leads to long access times or even application freezes, especially during garbage collection.

In general, the additional overhead caused by a larger heap is smaller than the gains in garbage collection frequency and allocation speed, as long as the heap does not get paged to disk. A good heap size setting is a heap that is as large as possible within the available physical memory.

There are two parameters for setting the heap size:

- `-Xms:`*`size`* sets the initial and minimum heap size.
- `-Xmx:`*`size`* sets the maximum heap size.

Example:

```
java -Xms:1g -Xmx:1g MyApplication
```

This command starts the JVM with a heap size fixed to 1 GB.

If the optimal heap size for the application is known, Oracle recommends that you set `-Xms` and `-Xmx` to the same value. This gives you a controlled environment where you get an appropriate heap size right from the start.

For more information, see the *Oracle JRockit Command-Line Reference*.

4.1.1.1 Setting the Heap Size on 64-bit Systems

On 64-bit systems a memory address is 64 bits long, which makes it possible to address much more memory than with a 32-bit address. Each reference also requires twice as much memory. To reduce the memory usage on 64-bit systems, the JRockit JVM can use compressed references. Compressed references reduce the references to 32 bits, and can be used if the entire heap can be addressed with 32 bits. Compressed references are enabled by default whenever applicable.

You can modify compressed references for various address bits and heap sizes by using the `-XXcompressedrefs` command-line option. For more information about `-XXcompressedrefs` and its parameters, see the *Oracle JRockit Command-Line Reference*.

4.1.2 Setting the Nursery and Keep Area Size

Command-line option: `-Xns:`*`size`*

The size of the nursery affects allocation speed, garbage collection frequency and garbage collection times. A small nursery fills quickly and must be garbage collected more often, while garbage collection of a large nursery takes slightly longer time. A nursery that is so small that few or no objects have died before a young collection is started is not useful, and neither is a nursery that is so large that no young collections are performed between garbage collections of the whole heap that are triggered due to allocation of large objects in old space.

An optimal nursery size for maximum application throughput is approximately half of the free heap. The goal is to have as many objects as possible be garbage collected by young collection rather than old collection. In the JRockit JVM, the dynamic garbage collection mode optimized for throughput, `-Xgc:throughput`, and the generational parallel garbage collector, `-Xgc:genpar`, dynamically sets the nursery size to an approximation of the optimal value.

The optimal nursery size for throughput is often quite large, which can lead to long young collection times. Because all Java threads are paused while the young collection is performed, consider reducing the nursery size below the optimal value to reduce the young collection pause times.

Example:

```
java -Xns:100m MyApplication
```

This command starts the JVM with a fixed nursery size of 100 MB.

For more information, see the *Oracle JRockit Command-Line Reference*.

4.1.2.1 Keep Area

Command-line option: `-XXkeepAreaRatio:`*`percentage`*

The keep area size affects both old collection and young collection frequency. A large keep area causes more frequent young collections, while a keep area that is too small causes more frequent old collections when objects are promoted prematurely.

An optimal keep area size is as small as possible while maintaining a low promotion ratio. The promotion ratio can be seen in the Flight Recorder recordings and verbose outputs from `-Xverbose:memory=debug`, and in the garbage collection report printed by `-Xverbose:gcreport`.

The keep area size is defined as a percent of the nursery.

Example:

```
java -XXkeepAreaRatio:10 MyApplication
```

This command starts the JVM with a keep area that is 10 percent of the nursery size.

For more information, see the *Oracle JRockit Command-Line Reference*.

4.2 Selecting and Tuning a Garbage Collector

The effect of garbage collection can be distributed in different ways depending on the choice of the garbage collection method. Use the command-line option `-Xgc:`*`mode`* to specify a garbage collection mode.

For more information about the `-Xgc` option, see the *Oracle JRockit Command-Line Reference*.

This section includes the following topics:

- Selecting a Garbage Collection Mode
- Tuning the Concurrent Garbage Collection Trigger

4.2.1 Selecting a Garbage Collection Mode

The garbage collection modes adjust the memory management system at run time, optimizing for a specific goal depending on the mode used.

The following are the garbage collection modes:

- **throughput**: Optimizes the garbage collector for maximum application throughput.
- **pausetime**: Optimizes the garbage collector for short and even pause times.
- **deterministic**: Optimizes the garbage collector for very short and deterministic pause times.

The garbage collection modes use advanced heuristics to tune the following parameters at run time:

- Nursery size
- Compaction amount and type

4.2.1.1 Throughput Mode

Command-line option: `-Xgc:throughput`

The throughput mode can either use a generational garbage collector (if `-Xgc:genpar` or only `-Xgc:throughput` is specified) or a nongenerational garbage collector (if `-Xgc:singlepar` or `-Xgc:parallel` is specified).

The `throughput` mode uses as little CPU resources as possible for garbage collection, giving the Java application as many CPU cycles as possible. The JRockit JVM achieves this by using a parallel garbage collector that stops the Java application during the entire garbage collection duration and uses all CPUs available to perform the garbage collection. Each individual garbage collection pause might be long, but in total, the garbage collector takes as little CPU time as possible.

Use the `throughput` mode for applications that demand a high throughput but are not very sensitive to the occasional long garbage collection pause.

The `throughput` mode is the default when the JVM runs in the `-server` mode (which is default) or can be enabled by using the `-Xgc:throughput` option.

Example:

```
java -Xgc:throughput MyApplication
```

This command starts the JVM with the garbage collection mode optimized for throughput.

For more information, see the *Oracle JRockit Command-Line Reference*.

4.2.1.2 Pausetime Mode

Command-line option: `-Xgc:pausetime`

The pausetime mode can use either a generational garbage collector (if `-Xgc:gencon` or only `-Xgc:pausetime` is specified) or a non-generational garbage collector (if `-Xgc:singlecon` is specified).

The `pausetime` mode aims to keeps the garbage collection pauses below a given pause target while maintaining as high a throughput as possible. The JRockit JVM achieves this with a mostly concurrent garbage collection strategy that allows the Java application to continue running during large portions of the garbage collection duration.

Use the `pausetime` mode for applications that are sensitive to long latencies (for example, transaction-based systems where transaction times are expected to be stable).

Example:

```
java -Xgc:pausetime MyApplication
```

This command starts the JVM with the garbage collection mode optimized for short pauses.

For more information, see the *Oracle JRockit Command-Line Reference*.

4.2.1.2.1 Setting a Pause Target for Pausetime Mode

Command-line option: `-XpauseTarget:`*`time`*

The `pausetime` mode uses a pause target for optimizing the pause times. The pause target affects the application throughput. A lower pause target inflicts more overhead on the memory management system.

Set the pause target as high as your application can tolerate.

Example:

```
java -Xgc:pausetime -XpauseTarget:300 MyApplication
```

This command starts the JVM with the garbage collection optimized for short pauses and a pause target of 300 milliseconds.

The default pause target for the `pausetime` mode is 500 milliseconds.

For more information, see the *Oracle JRockit Command-Line Reference*.

4.2.1.3 Deterministic Mode

Command-line option: `-Xgc:deterministic`

The `deterministic` mode ensures short garbage collection pause times and limits the total pause time within a prescribed window. The JRockit JVM achieves this by using a specially designed mostly concurrent garbage collector, which allows the Java application to continue running as much as possible during the garbage collection.

Use the `deterministic` mode for applications with strict demands on short and deterministic latencies (for example, transaction-based applications such as brokerage applications).

Example:

```
java -Xgc:deterministic MyApplication
```

This command starts the JVM with the garbage collection mode optimized for short and deterministic pauses.

For more information, see the *Oracle JRockit Command-Line Reference*.

4.2.1.3.1 Special Note for Oracle WebLogic Real Time Users

Deterministic garbage collection time can be affected by the JRockit Mission Control Client. While all JRockit Mission Control tools are fully supported when running Oracle WebLogic Real Time with the deterministic garbage collector, you should be aware of the following:

- The `-Xmanagement` option does not prolong deterministic garbage collection pauses by itself but it does introduce a slightly increased amount of Java code executed by the JVM. This can affect response times and performance compared to not using the `-Xmanagement` option.
- When making a recording by using the JRockit Flight Recorder, if you run the application in a latency-sensitive situation where you cannot accept the pause for the benefit of the information, disable heap statistics (`heapstat`). Heapstat provides additional bookkeeping of the content of the heap. These statistics are collected at the beginning and at the end of a Flight Recorder recording, inside a pause. You can disable heapstat by using specific arguments when requesting the recording.
- JRockit Flight Recorder recordings, even with heapstats disabled, might cause deterministic garbage collection pauses to last slightly longer.

- Memory leak trend analysis can cause longer garbage collection pauses, similar to JRockit Flight Recorder recordings.
- For requests for more information when the Memory Leak Detector is using its graphical user interface or the diagnostic command, a longer pause can be introduced (for example, to retrieve the number of instances of a type of object or to retrieve the list of references to an instance).

For more information about JRockit Mission Control, see the *Oracle JRockit Mission Control Online Help*.

4.2.1.3.2 Setting a Pause Target for Deterministic Mode

Command-line option: `-XpauseTarget:`*`time`*

The deterministic mode uses a pause target for optimizing the pause times. The pause target affects the application throughput. A lower pause target inflicts more overhead on the memory management system.

Set the pause target as high as your application can tolerate.

The garbage collector aims to keep the garbage collection pauses below the given pause target. Success of this strategy depends on the application and the hardware. For example, a pause target of 30 milliseconds has been verified on an application with 1 GB heap and an average of 30 percent of the live data or less at collection time, running on the following hardware:

- 2 x Intel Xeon 3.6 GHz, 2 MB level 2 cache, 4 GB RAM
- 4 x Intel Xeon 2.0 GHz, 0.5 MB level 2 cache, 8 GB RAM

Running on slower hardware, with a different heap size and with more live data can break the deterministic behavior or cause performance degradation over time, while faster hardware or less live data might allow you to set a lower pause target.

Example:

```
java -Xgc:deterministic -XpauseTarget:40ms MyApplication
```

This command starts the JVM with the garbage collection optimized for short and deterministic pauses and a pause target of 40 milliseconds.

The default pause target for the `deterministic` mode is 30 milliseconds.

For more information, see the *Oracle JRockit Command-Line Reference*.

4.2.1.4 Garbage Collector Mode Selection Workflow

To select the best garbage collection strategy for your application you can follow this workflow:

1. Is your application sensitive to long garbage collection pauses (500 milliseconds or more)?
   - **Yes**: Select a mostly concurrent garbage collection mode, `gencon` or `singlecon`.
   - **No**: Select a parallel garbage collection mode, `genpar` or `singlepar`.
2. Does your application allocate a lot of temporary objects?
   - **Yes**: Select a two-generational garbage collection mode, `gencon` or `genpar`.
   - **No**: Select a single-generational garbage collection mode, `singlecon` or `singlepar`.

For example, the Oracle WebLogic SIP Server is a transaction-based system that allocates new objects for each transaction and has short timeouts for transactions. Long garbage collection pauses would cause transactions to time out, so use a mostly concurrent garbage collection: `gencon` or `singlecon`. Because the transactions generate many temporary or short lived objects, a two-generational garbage collection mode (`gencon`) would be appropriate.

You can set a static garbage collection strategy with the `-Xgc:`*`mode`* option.

Example:

```
java -Xgc:gencon MyApplication
```

This command starts the JVM with the generational concurrent garbage collector.

4.2.1.5 Changing Garbage Collection Mode at Run Time

You can change the garbage collector mode at run time from the **Memory** tab of the JRockit Management Console (in JRockit Mission Control), except when using the `deterministic` and `singlepar` garbage collection modes.

For more information, see the *Oracle JRockit Mission Control Online Help*.

4.2.2 Tuning the Concurrent Garbage Collection Trigger

Command-line option: `-XXgcTrigger:`*`value`*

When you are using a concurrent strategy for garbage collection (in either the mark or the sweep phase, or both), the JRockit JVM dynamically adjusts when to start an old generation garbage collection in order to avoid running out of free heap space during the concurrent phases of the garbage collection. The triggering is based on such characteristics as how much space is available on the heap after previous collections. The JVM dynamically tries to optimize this space and, in doing so, it occasionally runs out of free heap during the concurrent garbage collection. When this happens, the following verbose output is displayed if the `-Xverbose:memdbg` option is used.

```
[DEBUG][memory ] [OC#55] Starting parallel sweeping phase.
[DEBUG][memory ] [OC#55] Will run parallel sweep due to: Alloc Queue Not Empty.
or
[DEBUG][memory ] [OC#55] Will run parallel sweep due to: Promotion Failed.
```

This message means that a concurrent sweep could not finish in time and the JVM is using all currently available resources to make up for it. In this case, a parallel sweep is made. If the JVM fails to adapt and the above output continues to appear, performance is affected.

To avoid this, set the `-XXgcTrigger` option to trigger a garbage collection when a specified percentage of the heap remains.

Example:

```
java -XXgcTrigger:20 MyApplication
```

This command triggers an old generation garbage collection when less than 20 percent of the free heap size is left unused.

If you are using a parallel garbage collection strategy (in both the mark and the sweep phase), old generation garbage collections are performed whenever the heap is completely full.

For more information, see the *Oracle JRockit Command-Line Reference*.

4.3 Tuning Compaction

Compaction is the process of moving chunks of allocated space towards the lower end of the heap, helping create contiguous free memory at the upper end. The JRockit JVM does partial compaction of the heap at each old collection. The size and position of the compaction area as well as the compaction method is selected by advanced heuristics, depending on the garbage collection mode used.

4.3.1 Fragmentation vs. Garbage Collection Pauses

Compaction of a large area with many objects increases the garbage collection pause times. On the other hand, insufficient compaction leads to fragmentation of the heap, which leads to lower performance. If the fragmentation increases over time, the JRockit JVM is eventually forced to either do a full compaction of the heap, causing a long garbage collection pause or throw an OutOfMemoryError.

If your application shows intermittent performance degradation over time, such that the performance degrades until it suddenly returns to normal, and then starts degrading again, you are most likely experiencing fragmentation problems. The heap becomes more and more fragmented for each old collection until, finally, object allocation becomes impossible and the JVM is forced to do a full compaction of the heap. The full compaction eliminates the fragmentation, but only until the next garbage collection. You can verify this by looking at the `-Xverbose:memory` outputs, monitoring the JVM through the Management Console in JRockit Mission Control, or by creating a JRockit Flight Recorder recording and examining the garbage collection data. If you see that the amount of used heap after each old collection keeps increasing over time until it peaks, and then decreases again at the next old collection, you are experiencing a fragmentation problem.

You can consider compaction to be optimally tuned when the fragmentation remains at a low level consistently.

4.3.2 Adjusting Compaction

Though the compaction heuristics in the JRockit JVM are designed to keep the garbage collection pauses low and consistent, you might sometimes want to limit the compaction ratio further to reduce the garbage collection pauses. In other cases, you might want to increase the compaction ratio to keep heap fragmentation in control.

You can adjust the compaction by using one of the following techniques:

- Setting the Compaction Ratio
- Setting Maximum References
- Turning Off Compaction
- Using Full Compaction

4.3.2.1 Setting the Compaction Ratio

Command-line option:`-XXcompaction:percentage=`*`value`*

Setting a static compaction ratio forces the JVM to compact a specified percentage of the heap at each old collection. This disables the heuristics for selecting a dynamic compaction ratio that depends on the heap layout. The compact ratio can be defined to a static percentage of the heap using the `-XXcompaction:percentage` option.

Example:

```
java -XXcompaction:percentage=5 MyApplication
```

This command starts the JVM with a static compact ratio of about 5 percent of the heap.

For more information, see the *Oracle JRockit Command-Line Reference*.

Use this option if you need to force the JVM to use a smaller or larger compaction ratio than it would select by default. You can monitor the compaction ratio in `-Xverbose:compaction` outputs and JRockit Flight Recorder recordings. A high compaction ratio keeps down the fragmentation on the heap but increases the compaction pause times.

4.3.2.2 Setting Maximum References

Command-line option: `-XXcompaction:maxReferences=`*`value`*

When compaction has moved objects, the references to these objects must be updated. The garbage collector does this before the Java threads are allowed to run again, which increases the garbage collection pause proportionally to the number of references that have been updated.

The maximum number of references that can exist from objects outside the compaction area to objects within the compaction area depends on the garbage collection mode used and, for some modes, is adjusted dynamically at run time.

To limit the pause caused by compaction, you can specify a static maximum number of references, by using the `-XXcompaction:maxReferences` option. If, during a garbage collection, the number of references to the chosen compaction area exceeds the specified `maxReferences` value, the compaction is cancelled.

Example:

```
java -XXcompaction:maxReferences=20000 MyApplication
```

This command starts the JVM with a maximum of 20000 references.

You can monitor the compaction behavior and compaction pause times in the `-Xverbose:compaction` output and Flight Recorder recordings. If you observe that many compactions are cancelled (skipped), increase the value of `maxReferences`; if the compaction pause times are too long, decrease the value of `maxReferences`.

For more information about maximum references, see the *Oracle JRockit Command-Line Reference*.

4.3.2.3 Turning Off Compaction

Command-line option: `-XXcompaction:enable=false`

Very few applications survive in the long run without any compaction at all but for those that do you can turn off the compaction entirely.

Example:

```
java -XXcompaction:enable=false MyApplication
```

For more information, see the *Oracle JRockit Command-Line Reference*.

4.3.2.4 Using Full Compaction

Command-line option: `-XXcompaction:full`

Some applications are not sensitive to garbage collection pauses or perform old collections very infrequently. For these applications, you might want to try running full compaction, as this maximizes the object allocation performance between the garbage collections. However, a full compaction of a large heap with a lot of objects can take several seconds to perform.

Example:

```
java -XXcompaction:full MyApplication
```

For more information, see the *Oracle JRockit Command-Line Reference*.

4.4 Optimizing Memory Allocation Performance

Apart from optimizing the garbage collection to clear space for object allocation, you can tune the object allocation itself to maximize the application throughput.

4.4.1 Setting the Thread Local Area Size

Command-line options:
`-XXtlaSize:min=`*`size`*`,preferred=`*`size`*`,wasteLimit=`*`size`*

The thread local area size influences the allocation speed, but can also have an impact on garbage collection frequency. A large TLA size allows each thread to allocate a lot of objects before requesting a new TLA, and in JRockit JVM it also allows the thread to allocate larger objects in the thread local area. On the other hand, a large TLA size prevents small chunks of free memory from being used for object allocation, which increases the impact of fragmentation. The TLA size is dynamic depending on the size of the available chunks of free space, and varies between a minimum and a preferred size.

Increasing the preferred TLA size is beneficial for applications where each thread allocates a lot of objects. When a two-generational garbage collection strategy is used, a large minimum and preferred TLA size also allow larger objects to be allocated in the nursery. Note however that the preferred TLA size should always be less than about 5 percent of the nursery size.

Increasing the minimum TLA size can improve garbage collection times slightly, as the garbage collector can ignore any free chunks that are smaller than the minimum TLA size.

Decreasing the preferred TLA size is beneficial for applications where each thread allocates only a few objects before it is terminated, so that a larger TLA would not ever become full. A small preferred TLA size is also beneficial for applications with very many threads, where the threads do not have time to fill their TLAs before a garbage collection is performed.

Decreasing the minimum TLA size lessens the impact of fragmentation.

A thread requests a new TLA earliest when the current TLA has memory less than the value specified by the `-XXtlaSize:wasteLimit` option. Decreasing the value of `-XXtlaSize:wasteLimit` makes better use of the available memory, but causes more 'slow-case' allocations in the memory other than TLA. A large value for the `-XXtlaSize:wasteLimit` option reduces slow allocations, but it can make garbage collections more frequent. Oracle recommends that you set `-XXtlaSize:wasteLimit` to the same value as `-XXtlaSize:min`.

A common setting for the TLA size is a minimum TLA size of 2 KB to 4 KB and a preferred TLA size of 16 KB to 256 KB.

Example:

```
java -XXtlaSize:min=1k,preferred=512k MyApplication
```

This command starts the JVM with a minimum TLA size of 1 KB and a preferred TLA size of 512 KB.

For more information, see the *Oracle JRockit Command-Line Reference*.

5 Tuning Locks

The interaction between Java threads affects the performance of your application. This chapter describes how to tune the interaction between Java threads.

There are two ways to tune the interaction of threads:

- By modifying the structure of your program code; for example, minimizing the amount of contention between threads
- By using options in the Oracle JRockit JVM that affect how contention is handled when your application is running

This chapter describes the following topics:

- Section 5.1, "Lock Profiling"
- Section 5.2, "Disabling Spinning Against Fat Locks"
- Section 5.3, "Adaptive Spinning Against Fat Locks"
- Section 5.4, "Lock Deflation"
- Section 5.5, "Lazy Unlocking"

For more information about how the JRockit JVM handles threads and locks, see the "Understanding Threads and Locks" section in *Oracle JRockit Introduction to the JDK*.

5.1 Lock Profiling

You can configure the JRockit Flight Recorder to collect and analyze information about the locks and the contention that occurred while the Flight Recorder was recording. To do this, add the following option when you start your application:

```
-XX:+UseLockProfiling
```

When lock profiling is enabled, you can view information about Java locks on the **Lock Profiling** tab in the JRockit Mission Control Client.

Note: Lock profiling creates significant (approximately 20 percent) overhead processing when your Java application runs.

Two diagnostic commands are tied to the lock profile counters. To work, both require lock profiling to be enabled with the `-XX:+UseLockProfiling` option. These are used with the `jrcmd` utility:

- The `lockprofile_print` command prints the current values of the lock profile counters.

- The `lockprofile_reset` command resets the current values of the lock profile counters.

For more information about diagnostic commands, see the *Oracle JRockit Diagnostics and Troubleshooting Guide*.

5.2 Disabling Spinning Against Fat Locks

Spinning against a fat lock is generally beneficial. However, in some instances such as when you have locks that create long waiting periods and high contention it can be expensive in terms of performance. You can turn off spinning against a fat lock and eliminate a potential performance degradation with the following option:

```
-XX:-UseFatSpin
```

The option disables the fat lock spin code in Java, enabling threads that are trying to acquire a fat lock to go to a sleep state directly.

5.3 Adaptive Spinning Against Fat Locks

You can let the JVM decide whether threads should spin against a fat lock or not (and directly go into sleeping state when failing to take it).

To enable adaptive lock spinning, use the `-XX:+UseAdaptiveFatSpin` option.

By default, adaptive spinning against fat locks is disabled. Note that whether threads that fail to take a particular fat lock spin or go to sleep can change at run time.

5.4 Lock Deflation

If the amount of contention on a fat lock is small, the lock converts to a thin lock. This process is called lock deflation. Thin locks have higher performance for uncontended locks. Therefore, lock deflation is enabled by default.

If you do not want fat locks to deflate, run the application with the following option:

```
-XX:-UseFatLockDeflation
```

With lock deflation disabled, a fat lock remains as a fat lock even after there are no threads waiting to take the lock.

You can also tune for when lock deflation is triggered. Specify, with the following option, the number of uncontended fat lock unlocks that occur before deflation:

```
-XX:FatLockDeflationThreshold=NumberOfUnlocks
```

5.5 Lazy Unlocking

Lazy unlocking is intended for applications with many nonshared locks. Note that it can introduce performance penalties with applications that have many short-lived but shared locks.

When lazy unlocking is enabled, locks are not released when a critical section is exited. Instead, once a lock is acquired, the next thread that tries to acquire such a lock must ensure that the lock is or can be released. It does this by determining if the initial thread is using the lock. A shared lock converts to a normal lock and does not stay in lazy mode.

Lazy unlocking is enabled by default in the JRockit JVM for all garbage collection strategies except the deterministic garbage collector.

6 Tuning for Low Latencies

This chapter describes how to tune the Oracle JRockit JVM for low latencies.

Long latencies can make some applications experience poor performance even though the overall throughput is acceptable. For example, a transaction-based system might seem to execute a satisfactory number of transactions during a specified amount of time but still show some uneven behavior with transactions occasionally timing out, even on low loads. Latencies in the application or the environment in which the application is run might cause this uneven or poor performance. Latencies occur due to anything from contention in the Java code to slow network connections to a database server. Latencies can also be caused by the JVM (during garbage collection for example) depending on how the JVM is tuned.

This chapter includes information about the following topics:

- Section 6.1, "Measuring Latencies"
- Section 6.2, "Tuning the Garbage Collector for Low Latencies"
- Section 6.3, "Tuning the Heap Size"
- Section 6.4, "Manually Tuning the Nursery Size"
- Section 6.5, "Manually Tuning Compaction"
- Section 6.6, "Tune when to Trigger a Garbage Collection"

6.1 Measuring Latencies

Application developers follow different methods to measure the performance of their application. You can, for example, run a set of simulated use cases and measure the time it takes to execute them, the number of a specific kind of transactions executed per minute, the average transaction time, or the percentage of the transaction time above or below a specific threshold. When you tune for low latencies, you are most interested in measuring the number of transaction times above a certain threshold. For best tuning results, you should have a varied set of benchmarks that are as realistic as possible and are run for a longer period of time. Twenty minutes is often a minimum; sometimes the full effect of the tuning can be seen only after several hours.

When you have identified a situation where the long latencies occur, you can start monitoring the JRockit JVM using some of the following methods:

- Create a run-time analysis report by using the JRockit Flight Recorder supplied with the product. If you are running JRockit JVM and Oracle JRockit Mission Control, the individual pause times for each garbage collection pause (there can be several pauses during one garbage collection) are reported. The JRockit Flight Recorder report also shows page faults occurring during garbage collection. For information about creating and analyzing a JRockit Flight Recorder report, see the *Oracle JRockit Mission Control Online Help*.

- You can create a latency recording to monitor the occurrences of latencies in your application. For more information about creating a latency recording, see the Oracle JRockit Mission Control Online Help.
- You can see garbage collection pause times in the JRockit JVM by starting the JVM with `-Xverbose:gcpause`.

6.2 Tuning the Garbage Collector for Low Latencies

The first step for tuning the JRockit JVM for low latencies is to select a garbage collection mode that gives you short garbage collection pauses. The options available are:

- Dynamic Garbage Collection Mode Optimized for Deterministic Pause Times

  This is the garbage collection mode designed for very short and deterministic garbage collection pauses. It is available as a part of Oracle JRockit Real Time.
- Dynamic Garbage Collection Mode Optimized for Short Pauses

  This is a garbage collection mode designed for short garbage collection pauses.
- Static Generational Concurrent Garbage Collection

  This static garbage collection mode provides fairly short garbage collection pauses but does not optimize for a specific pause target. Additional tuning of the nursery size and compaction might be necessary when this garbage collector is chosen.

For more information about garbage collector options, see Section 4.2, "Selecting and Tuning a Garbage Collector."

6.2.1 Dynamic Garbage Collection Mode Optimized for Deterministic Pause Times

Applications that require minimal latency, such as those used in the telecom and finance industries, cannot tolerate the unpredictable pause times caused by common garbage collection strategies. To avoid these long pause times, the JRockit JVM provides deterministic garbage collection, a dynamic garbage collection mode that keeps the garbage collection pauses short and deterministic.

Set the `deterministic` garbage collector as follows:

```
java -Xgc:deterministic -Xms:1g -Xmx:1g myApplication
```

The garbage collector try to keep the garbage collection pauses below the given pause target. How well it succeeds depends upon the application and the hardware. For example, a pause target of 30 milliseconds has been verified on an application with 1 GB heap and an average of 30 percent of the live data or less at collection time, running on the following hardware:

- 2 x Intel Xeon 3.6 GHz, 2 MB level 2 cache, 4 GB RAM
- 4 x Intel Xeon 2.0 GHz, 0.5 MB level 2 cache, 8 GB RAM

Running on slower hardware, with a different heap size and with more live data might break the deterministic behavior or cause performance degradation over time, while faster hardware or less live data might allow you to set a lower pause target.

The default pause target for the `deterministic` mode is 30 milliseconds, and can be changed with the `-XpauseTarget:`*`time`* option.

Example:

```
java -Xgc:deterministic -Xms:1g -Xmx:1g -XpauseTarget:40ms MyApplication
```

This command starts the JVM with the garbage collection optimized for short and deterministic pauses and a pause target of 40msec.

For more information, see the documentation on `-XpauseTarget`.

The deterministic garbage collector optimizes the compaction for the given pause target and does not use a nursery. Further tuning of compaction and nursery size should thus be unnecessary when the deterministic garbage collector is used.

6.2.2 Dynamic Garbage Collection Mode Optimized for Short Pauses

The dynamic garbage collection mode optimized for short pauses is useful for applications that don't require quite as short and deterministic pauses as the deterministic garbage collector guarantees. This garbage collection mode selects a garbage collection strategy to keep the garbage collection pauses below a given pause target (500 milliseconds by default). Compaction is also adjusted automatically to keep down the pause times caused by compaction.

Set the `pausetime` priority as follows:

```
java -Xgc:pausetime myApplication
```

If you use the `pausetime` priority but find that the default (500 milliseconds) is too long, you can specify a target pause time by using the `-XpauseTarget` option.

Example:

```
java -Xgc:pausetime -XpauseTarget=200ms myApplication
```

Note that there is a certain trade off between short pauses and application throughput. Shorter garbage collection pauses require more overhead in bookkeeping and can cause more fragmentation, which lowers the performance. If your application can tolerate pause times longer than 500 milliseconds you can increase the pause target to increase the application's performance.

The target value is used as a pause time goal and by the dynamic garbage collector to more precisely configure itself to keep pauses near the target value. Using this option enables you to specify the pause target to be between 200 milliseconds and 5 seconds. If you do not specify a pause target, the default remains 500 milliseconds.

The garbage collection mode for short pauses optimizes the compaction for the given pause target, so further tuning of the compaction should not be necessary. The nursery size is static for this garbage collection mode and has to be tuned manually.

6.2.3 Static Generational Concurrent Garbage Collection

If you want to use a static garbage collector and still experience minimal pause times, use a concurrent garbage collector. Generally, using a generational garbage collector is preferable to using a single-spaced garbage collector since a generational garbage collector gives you better application throughput.

To use a generational concurrent garbage collector, enter the following at the command line:

```
java -Xgc:gencon myApplication
```

To use a single-spaced concurrent garbage collector, enter the following at the command line:

```
java -Xgc:singlecon myApplication
```

When you use a static garbage collector, you might have to tune the nursery size and the compaction manually.

6.3 Tuning the Heap Size

You can resize the heap by using the `-Xms` (initial and minimum heap size) and `-Xmx` (maximum heap size) command-line options when you launch the JRockit JVM. Usually you can set the initial and the maximum heap size to the same value. Increasing the heap size reduces the frequency of garbage collections. A larger heap can also take slightly longer to garbage collect, but this effect is usually not considerable until the heap reaches sizes of several gigabytes.

The best approach to tuning the heap size is to benchmark the application with different heap sizes. Monitor the garbage collection pauses as described in Section 6.1, "Measuring Latencies" and determine the largest possible heap size for your application.

The only exception is the deterministic garbage collector. The deterministic garbage collector is verified to work best with a heap size of about 1 GB.

To set the heap size, use the `-Xms` and the `-Xmx` options.

Example:

```
java -Xms:1g -Xmx:1g myApplication
```

For more information, see Section 4.4, "Optimizing Memory Allocation Performance."

6.4 Manually Tuning the Nursery Size

If you are running `-Xgc:pausetime` or `-Xgc:gencon`, you might want to tune the nursery size manually.

The size of the nursery is static when you use `-Xgc:pausetime`, but setting it manually results in a more even behavior. For more information about the default nursery size, see the documentation for -Xns in *Oracle JRockit Command-Line Reference*. Note that when you use the dynamic garbage collector, the nursery might also be turned off completely when single-spaced garbage collection is used.

Note: Tuning the nursery size manually is often beneficial for both the pause times and the application throughput.

The default nursery size for `-Xgc:gencon` is static, and it might not be optimal for all applications. You might benefit from manually setting a custom nursery size. The nursery should be as large as possible, but the nursery size must be decreased if the pause time created by a young collection (nursery garbage collection) is too long. Tune the nursery size by benchmarking your application with several different nursery sizes while monitoring the garbage collection pauses as described in Section 6.1, "Measuring Latencies."

To set the size of the nursery, use the `-Xns` option.

Example:

```
java -Xgc:pausetime -Xns:64m myApplication
```

6.5 Manually Tuning Compaction

If you are using a static garbage collector, tuning compaction manually can help improve latencies. Compaction is performed during a garbage collection pause, and thus the compaction time affects the garage collection pause times. By default, the static garbage collectors use a compaction scheme that aims at keeping the compaction times fairly even, but does not put an upper bound on the compaction time.

You can limit the compaction manually by setting a static compaction area size (`-XXcompaction`) or by limiting the number of references that can be updated due to compaction (`-XXcompaction:maxReferences`). These actions does not guarantee an upper bound on the compaction time, but reduce the risk for long compaction times.

Note that if you set the compaction ratio to low, the heap slowly becomes more and more fragmented until it is impossible to find free space that is big enough for object allocation. The heap becomes full of *dark matter* (basically severe fragmentation). When this happens, a compaction of the complete heap is done, which can result in a pause time of up to half a minute. Dark matter is reported for the heap in a Flight Recorder recording.

For more information about limiting compaction, see Section 4.3.2, "Adjusting Compaction."

6.6 Tune when to Trigger a Garbage Collection

The `-XXgcTrigger` option determines how much free memory should remain on the heap when a concurrent garbage collection starts. If the heap becomes full during the concurrent garbage collection, the Java application cannot allocate more memory until garbage collection frees some memory, which might cause the application to pause. While the trigger value tunes itself at run time to prevent the heap from becoming too full, this automatic tuning might take too long. Instead, you can use `-XXgcTrigger` option to set, from the start, a garbage collection trigger value more appropriate to your application.

If the heap becomes full during the concurrent mark phase, the sweep phase reverts to parallel sweep (unless `-XXnoParSweep` has been specified). If this happens frequently and the garbage collection trigger does not increase automatically to prevent this, use `-XXgcTrigger` to manually increase the garbage collection trigger.

Example:

```
java -XXgcTrigger myApp
```

The current value of the garbage collection trigger appears in the `-Xverbose:memdbg` output whenever the trigger changes.

7 Tuning for Better Application Throughput

This chapter describes how to tune the JRockit JVM for improved application throughput.

Every application has unique behavior and has unique requirements on the JVM for gaining maximum application throughput. The standard behavior of the Oracle JRockit JVM gives good performance for most applications. However, you can tune the JVM further to increase application throughput.

This chapter includes information about the following topics:

7.1 Measuring Application Throughput

In this document *application throughput* denotes the speed at which a Java application runs. If your application is a transaction-based system, high throughput means that more transactions are executed during a given amount of time. You can also measure the throughput by measuring how long it takes to perform a specific task or calculation.

To measure the throughput of your application you need a benchmark. The benchmark should simulate several realistic use cases of the application and run long enough to allow the JVM to warm up and perform several garbage collections. You also need a way to measure the results, either by timing the entire run of a specific set of actions or by measuring the number of transactions that can be performed during a specific amount of time. For an optimal throughput assessment, the benchmark should run on high load and not depend on any external input like database connections.

When you have a benchmark set up, you can monitor the behavior of the JVM using one of the following methods:

- Create a run-time analysis with the JRockit Flight Recorder in Oracle JRockit Mission Control. In the Flight Recorder tool, you can see the frequency of the garbage collections and why garbage the collections are launched. This information provides clues for memory management tuning. For information about creating and analyzing a Flight Recorder report, see the *Oracle JRockit Mission Control Online Help*.

- Create verbose outputs by using the `-Xverbose` option. For example, the `-Xverbose:memdbg,gcpause,gcreport` option causes memory management data such as garbage collection frequency and duration to be displayed. The `-Xverbose:memdbg` option also causes the reason for garbage collection to be displayed. This helps you study the garbage collection behavior.

Now you have the tools for measuring the throughput of your Java application and can start to tune the JVM for better application throughput.

7.2 Selecting the Garbage Collector for Maximum Throughput

The first step of tuning the JRockit JVM for maximum application throughput is to select an appropriate garbage collection mode or strategy. The options available are:

- Dynamic Garbage Collection Mode Optimized for Throughput

  This is the default garbage collection mode for the JRockit JVM. This mode selects the optimal garbage collection strategy for maximum application throughput.

- Static Generational Parallel Garbage Collection

  This static garbage collector is a good alternative if you do not want to use a dynamic garbage collection mode. The generational parallel garbage collector provides high throughput for applications that allocate a lot of temporary objects.

- Static Single-Spaced Parallel Garbage Collection

  This is another alternative if you do not want to use a dynamic garbage collection mode. The single-spaced parallel garbage collector provides high throughput for applications that allocate mostly large objects.

For more information about garbage collector options, see Section 4.2, "Selecting and Tuning a Garbage Collector."

7.2.1 Dynamic Garbage Collection Mode Optimized for Throughput

The default garbage collection mode in the JRockit JVM (assuming that you run in server mode, which is also default) tunes the memory management for maximum application throughput. By default, it selects a generational garbage collection strategy. It also tunes the nursery size, if the garbage collection strategy is generational.

Note that if you use the dynamic garbage collection mode optimized for throughput, the garbage collection pauses do not have any strict time limits. If your application is sensitive to long latencies, you should tune for low latencies rather than for maximum throughput, or find a middle path that gives you acceptable latencies.

The dynamic garbage collection mode optimized for throughput is the default garbage collector for the JRockit JVM. You can also turn it on explicitly as follows:

```
java -Xgc:throughput myApplication
```

7.2.2 Static Single-Spaced Parallel Garbage Collection

If you want to use a static garbage collector, then you should use a parallel garbage collector in order to maximize application throughput. If the large or small object allocation ratio is high, then use a single-spaced garbage collector (`-Xgc:singlepar`). You can see the ratio between large and small object allocation if you do a Flight Recorder recording of your application.

7.2.3 Static Generational Parallel Garbage Collection

If you want to maximize application throughput and the large or small object allocation ratio is low, then use a generational parallel garbage collector (`-Xgc:genpar`). A generational parallel garbage collector might be the right choice even if the large or small object allocation ratio is high when you are using a very small nursery. You can see the ratio between large and small object allocation if you do a Flight Recorder recording of your application.

7.3 Tuning the Heap Size

The default heap size starts at 64 MB and can increase up to 3 GB (for 64-bit systems) or 1 GB (for 32-bit systems). Most server applications need a large heap, at least larger than 1 GB, to optimize throughput. For such applications, you should set the heap size manually by using the `-Xms` (initial heap size) and `-Xmx` (maximum heap size) command-line options. Setting `-Xms` the same size as `-Xmx` has regularly shown to be the best configuration for improving throughput.

Example:

```
java -Xms:2g -Xmx:2g myApplication
```

For more information about setting the initial and maximum heap sizes, including guidelines for setting these values, please see Section 4.4, "Optimizing Memory Allocation Performance."

7.4 Manually Tuning the Nursery Size

The nursery or young generation is the area of free chunks in the heap where objects are allocated when running a generational garbage collector (`-Xgc:throughput`, `-Xgc:genpar` or `-Xgc:gencon`). A nursery is valuable because most objects in a Java application die young. Collecting garbage from the young space is preferable to collecting the entire heap, as it is a less expensive process and most objects in the young space would already be dead when the garbage collection is started.

If you are using a generational garbage collector you might need to change the nursery setting to accommodate more young objects.

- `-Xgc:throughput` and `-Xgc:genpar` change the nursery size dynamically at run time. In some cases manual tuning can result in a more efficient nursery size.
- `-Xgc:gencon` has a fairly low and static nursery size setting. For many applications, you might want to tune the nursery size manually when using this garbage collector.

An efficient nursery size is such that the amount of memory freed by young collections (garbage collections of the nursery) rather than old collections (garbage collections of the entire heap) is as high as possible. To achieve this, you should set the nursery size close to the size of half of the free heap after an old collection.

To set the nursery size manually, use the `-Xns` option.

Example:

```
java -Xgc:gencon -Xms:2g -Xmx:2g -Xns:512m myApplication
```

7.5 Manually Tuning Compaction

Compaction is the process of moving chunks of allocated space toward the lower end of the heap, helping to create contiguous free memory at the other end. The JRockit JVM does partial compaction of the heap at each old collection.

The default compaction setting for garbage collectors (`-Xgc`) uses a dynamic compaction scheme that tries to avoid peaks in the compaction times. This is a compromise between keeping garbage collection pauses even and maintaining a good throughput, so it does not necessarily give the best possible throughput. Tuning the compaction can pay off well, depending on the characteristics of the application.

There are two ways to tune the compaction for better throughput; increasing the size of the compaction area and increasing the maximum references. Increasing the size of the compaction area helps to reduce the fragmentation on the heap. Increasing the maximum references implicitly allows larger areas to be compacted at each garbage collection. This reduces the garbage collection frequency and makes allocation of large objects faster, thus improving the throughput.

For information about tuning these compaction options, see Section 4.3, "Tuning Compaction."

7.6 Tuning the Thread-Local Area Size

Increasing the preferred TLA size speeds up allocation of small objects when each Java thread allocates a lot of small objects, because the threads do not have to synchronize to get a new TLA as often.

In Oracle JRockit JVM, the preferred TLA size also determines the size limit for objects allocated in the nursery. Increasing the TLA size also allows larger objects to be allocated in the nursery, which could be beneficial for applications that allocate a lot of large objects. A JRockit Flight Recorder recording provides you with statistics on the sizes of large objects allocated by your application. For good performance, you can try setting the preferred TLA size at least as large as the largest object allocated by your application.

For information about how to set the TLA size, see Section 4.4.1, "Setting the Thread Local Area Size."

8

Tuning for Stable Performance

This chapter describes how to tune the JVM for stable performance.

An incorrectly tuned JVM might perform well initially, but could start showing lower performance or longer latencies over time or display severe performance variations.

This chapter includes the following topics:

8.1 Measuring the Performance Variance

To measure and analyze performance variance over time you need a long-running test that continuously reports the current performance. The test scenario should be as realistic as possible and cover as many use cases as possible.

When you have identified a variance in performance you can start monitoring the Oracle JRockit JVM to check whether the variance correlates to events within the JVM (for example, garbage collection, fragmentation, or lock deflation). The tools in Oracle JRockit Mission Control and the verbose outputs generated when you use the `-Xverbose` option help you analyze the variance.

Table 8–1 lists the events to look for and the tools to detect and analyze each event.

Table 8–1 JVM Events

Event Type	What to Look For	Tools for Analysis
Heap size change	The heap increases or decreases	`-Xverbose:memdbg`, JRockit Flight Recorder, Oracle JRockit Mission Control
Nursery size change	The nursery size increases or decreases	`-Xverbose:memdbg`, JRockit Flight Recorder, Oracle JRockit Mission Control
Garbage collector strategy change	A dynamic garbage collection mode changes the garbage collection strategy	`-Xverbose:memdbg`, JRockit Flight Recorder
Increased fragmentation	The amount of dark matter increases	JRockit Flight Recorder

Table 8–1 (Cont.) JVM Events

Event Type	What to Look For	Tools for Analysis
Full compaction	Compaction of all heap parts at once	`-Xverbose:memdbg`, JRockit Flight Recorder

8.2 Tuning the Heap Size

Heap size changes at run time can cause performance variations. You can monitor the heap size in `-Xverbose:memdbg` outputs and in JRockit Mission Control tools. A JRockit Flight Recorder recording also indicates whether the heap size changed during the recording.

For an even performance over time, you should set the initial heap size (`-Xms`) to the same value as the maximum heap size (`-Xmx`).

Example:

```
java -Xms:1g -Xmx:1g myApplication
```

For more information about tuning the heap size, see Section 4.1.1, "Setting the Heap Size."

8.3 Manually Tuning the Nursery Size

Nursery size changes at run time can cause performance variations, but can also help keeping the performance high when the load changes. You can monitor the nursery size in `-Xverbose:memdbg` outputs and JRockit Mission Control tools. A Flight Recorder recording also indicates whether the nursery size changed during the recording.

If you find that performance variations in your application correlate to nursery size changes, you can set a static nursery size by using the `-Xns` option.

Example:

```
java -Xns:100m myApplication
```

For more information about tuning the nursery size, see Section 4.1.2, "Setting the Nursery and Keep Area Size."

Note: Overriding the dynamic nursery sizing heuristics can have a negative impact on the performance or cause performance variations in applications where the amount of live data varies during the run.

8.4 Tuning the Garbage Collector

The garbage collection modes in the JRockit JVM select a garbage collection strategy based on run-time information. Changes in application behavior can cause the garbage collection mode to change. If such changes happen often and cause performance variations, you might want to select a different garbage collection mode. You can change the garbage collection mode by using the `-Xgc:`*`mode`* option.

Example:

```
java -Xgc:genpar myApplication
```

For more information, see Section 4.2.1, "Selecting a Garbage Collection Mode."

8.5 Tuning Compaction

The Oracle JRockit JVM uses the mark-and-sweep garbage collection model (for more information, see the "Mark-and-Sweep Model" section in *Oracle JRockit Introduction to the JDK*).

This mark-and-sweep garbage collection model could cause the heap to become fragmented, which means that the free areas on the heap increase in number but become small. The JVM performs partial compaction of the heap at each garbage collection to reduce the fragmentation. Sometimes, the amount of compaction is not enough. This leads to increasing fragmentation, which, in turn, leads to more and more frequent garbage collections until the heap is so fragmented that a full compaction is performed. After the full compaction, the garbage collection frequency decreases, but gradually increases as the fragmentation increases again.

This behavior causes the performance of the Java application to vary. As the garbage collection frequency increases, the performance drops. During the full compaction, you might experience a prolonged garbage collection pause, which pauses the entire Java application for a while. After this, the performance is high again, but starts going down as the garbage collection frequency increases again.

You can monitor the compaction ratio and garbage collection frequency in `-Xverbose:memdbg` outputs, the Management Console, and Flight Recorder recordings. A Flight Recorder recording also shows you how much dark matter (severe fragmentation) exists on the heap. If you find that the garbage collection keeps increasing until a full compaction is done, you need to increase the compaction ratio. For information about how to tune compaction, see Section 4.3, "Tuning Compaction."

You can also decrease the fragmentation on the heap by using a generational garbage collector. For more information, see Section 4.2, "Selecting and Tuning a Garbage Collector."

9 Tuning for a Small Memory Footprint

This chapter describes the tuning options available to reduce the memory footprint of the JVM.

If you are running on a system with limited memory resources, consider tuning the Oracle JRockit JVM for a small memory footprint.

This chapter includes information about the following topics:

9.1 Measuring the Memory Footprint

The memory footprint of an application can be measured by using some of the tools available in the operating system (for example, the top shell command or the Task Manager in Windows).

To determine how the memory usage of the JVM process is distributed, request a memory analysis by using the `jrcmd` command to print the memory usage of the JVM. For more information, see *Oracle JRockit JDK Tools*.

After you get information about the memory usage of the JVM, you can start tuning the JVM to reduce the memory footprint within the areas that use the most memory.

9.2 Setting the Heap Size

The most obvious place to start tuning the memory footprint is the Java heap size. If you reduce the Java heap size, you reduce the memory footprint of the Java process by the same amount. You cannot reduce the heap size beyond a point; the heap should be large enough for all objects that are live at the same time. Preferably, the heap must be at least twice the size of the total amount of live objects, or large enough so that the JVM spends less time garbage collecting the heap than running Java code.

You can set the heap size by using the `-Xms` (initial heap size) and `-Xmx` (maximum heap size) options.

Example:

```
java -Xms:100m -Xmx:100m myApplication
```

To let the heap to grow and shrink depending on the amount of free memory in your system, set the`-Xms` command lower than the `-Xmx` command. For more information about setting the heap size, see Section 4.4, "Optimizing Memory Allocation Performance."

9.3 Selecting a Garbage Collector

The choice of a garbage collection mode or static strategy does not in itself affect the memory footprint noticeably, but if you choose the right garbage collection strategy, you reduce the heap size without a major performance degradation.

If your application uses a lot of temporary objects, consider using a generational garbage collection strategy. A nursery reduces fragmentation and allows for a smaller heap.

The concurrent garbage collector must start garbage collections before the heap is entirely full, to allow Java threads to continue allocating objects during the garbage collection. This means that the concurrent garbage collector requires a larger heap than the parallel garbage collector, and thus your primary choice for a small memory footprint is a parallel garbage collector.

The default garbage collection mode is a generational parallel garbage collection strategy. This means that the default garbage collector is a good choice when you want to minimize the memory footprint.

To change the garbage collection mode, specify the mode by using the `-Xgc` option.

Example:

```
java -Xgc:genpar myApplication
```

For more information about selecting a garbage collector, see Section 4.2, "Selecting and Tuning a Garbage Collector."

9.4 Tuning Compaction

Using a small heap increases the possibility of fragmentation on the heap. Fragmentation can have a severe effect on application performance, both by lowering the throughput and by causing occasional long garbage collections when the garbage collector is forced to compact the entire heap at once.

If you experience problems with fragmentation on the heap, increase the compaction ratio by using the `-XXcompaction:percentage` option.

Example:

```
java -XXcompaction:percentage=20 myApplication
```

If your application is not sensitive to long latencies, consider using full compaction. Full compaction enables you to use a smaller heap, because all fragmentation is eliminated at each garbage collection. You can enable full compaction by using the `-XXcompaction:full` option.

Example:

```
java -XXcompaction:full myApplication
```

Compaction uses memory outside of the heap for bookkeeping. As an alternative to increasing the compaction, use a generational garbage collector, which also reduces the fragmentation.

9.5 Tuning Object Allocation

You can tune the object allocation to allow smaller chunks of free memory to be used for allocation. This reduces the negative effects of fragmentation, and allows you to run with a smaller heap. The smallest chunk of memory used for object allocation is a thread local area. Free chunks smaller than the minimum thread local area size are ignored by the garbage collector and become *dark matter* until a later garbage collection frees some adjacent memory or compacts the area to create larger free chunks.

You can reduce the minimum thread local area size by using the `-XXtlaSize:min=size` option.

Example:

```
java -XXtlaSize:min=1k myApplication
```

For more information about how to set the thread local area size, see the documentation on `-XXtlaSize` and Section 4.4, "Optimizing Memory Allocation Performance."

10 Tuning for a Faster JVM Startup

This chapter describes how to tune the JVM to decrease the startup time.

Small utility applications that run only for a short time might have a degraded performance if the JVM and Java application startup time is long. The Oracle JRockit JVM is by default optimized for server use, which means that the startup times can be longer in favour of high performance as soon as the application is up and running.

This chapter includes information about the following topics:

- Section 10.1, "Measuring the Startup Time"
- Section 10.2, "Setting the Heap Size"
- Section 10.3, "Troubleshooting Your Application and the JVM"

10.1 Measuring the Startup Time

The startup time of an application is the time it takes for the application to start running, and be ready to perform its designated activities. The startup time includes both the JVM startup and the Java application startup times.

For information about how to measure the startup time of your application, see the *Oracle JRockit Diagnostics and Troubleshooting Guide*.

10.2 Setting the Heap Size

The heap size affects both the JVM startup time and the Java application startup time. The JVM reserves memory for the maximum heap size (`-Xmx`) and commits memory for the initial heap size (`-Xms`) at startup, which takes time. For large applications, this is inevitable. Note that using an oversized heap can lead to longer than necessary JVM startup times. If your application is small and runs for only a short time, you might have to set a small heap size to avoid the overhead of reserving and committing more memory than the application requires.

If the initial heap is too small, the Java application startup becomes slow as the JVM is forced to perform garbage collection frequently until the heap grows to a more reasonable size. For optimal startup performance, set the initial heap size to the same as the maximum heap size.

10.3 Troubleshooting Your Application and the JVM

The application can cause the slow startup. For information about troubleshooting problems in the application and JVM, see the *Oracle JRockit Diagnostics and Troubleshooting Guide*.